Model-driven Development of Evolving Secure Software Systems

Sven Peldszus
University of Koblenz-Landau
Koblenz, Germany
speldszus@uni-koblenz.de

***Abstract*—Software systems are continuously entering more and more parts of our lives and have to deal with a higher amount of sensitive data than ever before. At the same time, these software systems get more complex and have to be maintained over long periods. One approach to deal with the issues arising from these trends is model-driven software development (MDD). Much research has been done on automating MDD approaches and integrating the different artifacts used. However, there are still plenty of issues that have to be solved. In this work, we discuss the three main challenges we discovered at the development of our MDD approach GRAViTY. GRaViTY supports developers in the model-driven development and maintenance, and evolution of long-living secure software systems. Thereby, GRaViTY itself leverages multiple MDD approaches.**

***Index Terms*—model-driven development, software engineering, evolution, maintenance, security**

I. Introduction

Modern software systems tend to be used on a long-term basis in environments prone to changes, are highly interconnected, are continuously extended with new features, and often process security-critical data [1], [2], [3]. These trends complicate to keep up with the ever-changing security precautions, attacks, and mitigations, which is vital for preserving a system's security. Model-driven development (MDD) enables us to address security issues in the early phases of the software design already, such as in UML models defined at design time [1]. Unfortunately, the specification of a system's security assumptions and documentation is often inconsistent with its implementation [4]. The continuous changes in the security assumptions and the design of software systems – for instance, due to structural decay [5] – have to be reflected in both the system models (e.g., UML models) and the system's implementation (including program models used, e.g., for static analysis or verification).

The tracing between the different artifacts available for deciding which change is necessary at which location in the system and on which of the many artifacts, has currently to be performed manually by developers. The effort for the creation of such mappings after the fact is still high even if this process is guided by tool support, e.g., for the creation of mappings between models and code [4]. For this reason, we have to maintain mappings between different artifacts used in the different phases of development from the very early beginning and to automate them as much as possible.

To tackle these challenges, we started to develop the GRaViTY framework [6]. This framework allows us to automatically create and maintain trace links between different artifacts, such as UML models, Java source code, and program models for analyses, created during the development of a system. Starting from early design-time models until the creation and maintenance of the code, this framework is intended

1) to maintain trace links between these artifacts,
2) keep all artifacts up to date if one artifact is changed,
3) to specify security requirements on the most suitable representation of a system, and
4) to continuously check all system representations for security violations.

For solving this challenge, we mainly utilize bidirectional graph transformation approaches. In this work, we are giving an overview of our solutions and discuss the three main challenges we faced:

1) Transformation between models with different granularity
2) Incremental updates of abstract syntax trees (AST)
3) Maintaining networks of transformations

In what follows, we introduce the relevant background on our assumptions about MDD as well as on security checks in Sec. II. Afterward, in Sec. III, we give a brief overview of the GRaViTY framework. In Sec. IV we discuss challenges we faced, and present our solutions and challenges that have to be overcome. How others dealt with the same challenges, is discussed in Sec. V. Finally, we conclude in Sec. VI.

II. Background

In this work, we present an approach for supporting developers in the model-driven software development (MDD) of secure software systems. For explaining our approach and the challenges, that we identified during its implementation, in this section, we introduce the underlying understanding of MDD and which artifacts we use as well as the different security checks which are combined by our approach for enforcing the development of a secure system.

A. Model-driven Software Development

In this work, we are building upon the concept of model-driven software development (MDD) [7]. MDD allows developers to specify the system and its properties on a higher level of abstraction than the source code level [8]. Thereby,

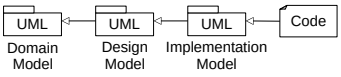


Fig. 1: Artifacts used in Model-driven Software Development

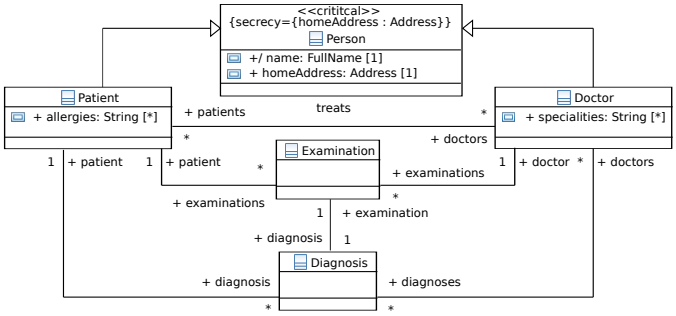


Fig. 2: Excerpt of a Hospital Domain Model based on [11]

developers should specify all additional properties, like security assumptions, only once on the most suitable level of abstraction. While MDD can cover many kinds of models we focus on UML models [9].

In GRaViTY, the single models are iteratively refined until we reach a concrete implementation of the system. Fig. 1 shows the refinement hierarchy of the model kinds currently considered by us from the most abstract model at the left to the final implementation at the right. All in all, we consider UML models with three different levels of granularity.

a) Domain Model: The most abstract model is a domain model, specifying general properties of the domain, that the software to develop is placed in [10]. Domain models are used in the earliest phases of software development to capture general properties about a system's domain. Often, domain models are specified using UML class diagrams.

Fig. 2 shows an excerpt of a domain model for hospitals. In hospitals, two kinds of people play a central role, patients and doctors who treat the patients. Both have a `name` and `homeAddress`. For patients, usually, a list of `allergies` is stored and for doctors a list of their `specialties`. A doctor can examine a patient in an `Examination` and create `Diagnose` in such examinations.

When we implement a software system for a hospital, e.g., like iTrust [12] for online management of patient data, we have to support the concepts captured in the domain model.

b) Design Model: After the specification of the domain model, the domain elements realized in the software are concretized in design models. Those design models specify the design of the system and how the functionality is distributed among the system. Thereby, the foundation of an easily maintainable system is set by the appropriate use of well-known design patterns [13]. This is also the first point where we have to start to continuously use design and security analyses to ensure the system's maintainability and security.

Fig. 3 shows an excerpt of a design model for iTrust, based on a UML model reverse-engineered by Bürger at al. [3]. In this model, different controls are specified for using the iTrust

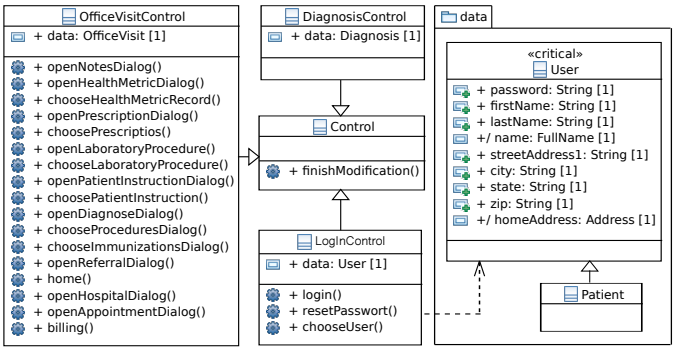


Fig. 3: Excerpt of a Design Model for iTrust based on [3]

platform, e.g., a login control, a control for documenting an office visit or for entering a diagnosis, as well as a more detailed data structure than in the domain model.

The different controls specify essential actions that can be performed, e.g., the option to reset the password in the login window. For a login of a user, the system needs the user information to identify and legitimate the user. For this purpose, the `LogInControl` accesses the data available in the `User`-object given to it.

The data used by the system is detailed in this model. For example, the classes `User` and `Patient` can be seen as more concrete instances of the classes `Person` and `Patient` from the domain model in Fig. 2. On the `Person` class, for example, it is explicitly specified that the `homeAddress` attribute, already known from the domain model, is derived from other attributes.

While models with different abstractions are often created separately, we encourage developers to model the information of refinement explicitly by the use of inheritance relations between elements. For example, there should be an explicit inheritance relationship between the `User` in the design model and the `Person` in the domain model.

c) Implementation Model: Precise functionality is specified in an implementation model. The implementation model is usually the first platform-dependent model and contains information about the deployment or languages used to implement the system. The implementation model can directly be executed, used for code generation, implemented manually, or a combination of all. In our approach, we support a combination of the code generation and manual implementation.

Fig. 4 shows an excerpt of an implementation model showing how the iTrust platform could be developed in a hospital. The model is based on our experience in modeling the hospital system of a partner in the VisiOn EU project together with them but does not show a real system [14].

Inside of the hospital, two servers are operated, one running iTrust and one running a database as well as an authentication service. Doctors are accessing the iTrust system from the hospital's local network. Patients can get access to their data from the outside but have to authenticate themselves at the authentication service provided by the hospital.

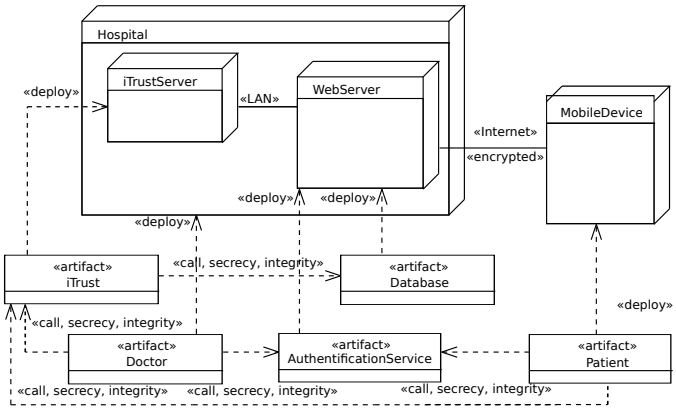


Fig. 4: Excerpt of an Implementation Model for iTrust based on [14]

B. Security Checks

For the enforcement of security requirements, we can make use of various kinds of security checks, supported by GRaViTY. In what follows, we give a brief overview of different kinds of security checks and in which stages of model-driven software development they can be applied.

1) Model-based Checks: According to the principle of security by design, the system to be developed should already be checked early during its development for security issues.

The UMLsec [1] approach, integrated into GRaViTY, allows the specification and check of essential security requirements already at design time. In UMLsec, UML models are annotated with security requirements like security levels of class members. These security annotations are checked for their compliance with different security policies.

In the given example, the class `Person` from the domain model in Fig. 2 is annotated with the UMLsec stereotype `<<critical>>`, which specifies that the attribute `homeAddress` is on the security level *secrecy*, meaning that only legitimate entities are allowed to read its value. UMLsec allows, as part of the `<<secure dependency>>` security policy, to check if this domain model or any model refining the domain model contains insecure uses of attributes or operations, that are annotated with a security requirement.

Here, we can utilize the use of refinement relations between the different model kinds for detecting security violations at no additional cost for considering multiple models. Also, if a security requirement is changed in one representation we can immediately see the impact on the other UML representations.

In the implementation model, we also annotated the calls and communication paths with UMLsec stereotypes. E.g., all data transferred from and to the doctors is sent over an internal LAN connection and all data sent from and to the patients is sent over an encrypted internet connection.

2) Static Code Analysis: Static code analysis is usually used to detect security issues during software implementation. Thereby, the analysis tools are often integrated within the development environments or build processes.

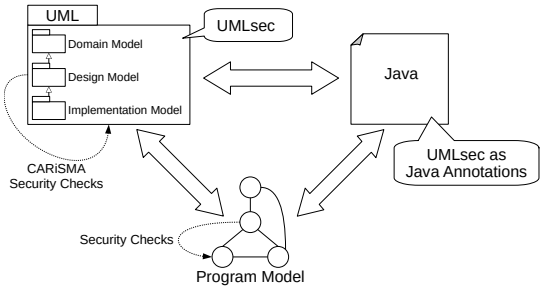


Fig. 5: Concept of the Framework

a) Analysis of API calls: Many approaches locally analyze calls to critical APIs and whether the chosen parameters have been selected securely. This covers, for example, calls to crypto APIs [15] or SQL queries [16]. While those approaches are important for the development of secure systems, in this work we are focusing more on the question whether, e.g. the use of a crypto API, has been implemented at a point specified in the models.

b) Secure data flow analysis: A common approach to detect leaks of secret data is a secure data flow analysis. One of the main problems for a precise data flow analysis is the classification of critical sources and sinks. Many tools are based on shared libraries of well known critical sources and sinks, created manually or by machine learning [17]. However, more precise information, especially about critical sources, is available in design-time models, e.g., annotated with UMLsec. For example, in Fig. 2 we declared the property `homeAddress` to contain secret values, which has to be considered during a secure data flow analysis.

While all these different security checks on the different artifacts can help in the development of a secure system, they are often limited to their area of focus. However, such security checks are more powerful when they are combined. For example, often information required by a security check on a lower level has already been defined at design-time. This information should be reused to avoid misunderstandings and divergence in the security assumptions but also to improve the effectiveness of the checks. Unfortunately, doing so is challenging and should be assisted by tool support.

III. GRaViTY Framework

Our proposed framework, called GRaViTY [6], supports developers in applying the model-driven development approach, as described in Sec. II-A, to the development and maintenance of secure long-living systems. As shown in Fig. 5, design models (e.g. specified in UML), source code (e.g. written in Java), and a program model for performing sophisticated analyses, e.g. the security checks from Sec. II-B, are continuously synchronized for covering the different phases of software development.

The program model provides a high-level abstraction from the pure Java source code [18], e.g., by reducing details from the statement level to access edges between the single members. In addition, easy to query structures are created,

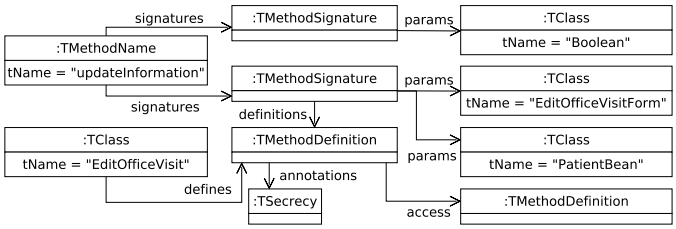


Fig. 6: Excerpt from the iTrust Program Model

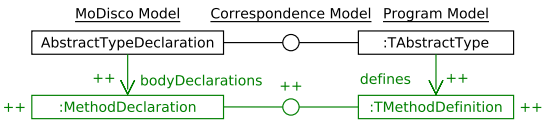
such as structuring methods and fields into a tree with names, signatures, and definitions. For example, Fig. 6 shows a program model with two different method signatures for the method name `updateInformation`. For the signature with the parameter types `EditOfficeVisitForm` and `Boolean`, a definition from the class `EditOfficeVisit` is shown, which calls an other method definition. This allows the easy specification of, e.g., refactorings [19], [18], anti-pattern detection [2] and elimination [20], or compliance checks with models [4].

Security-related specifications are introduced into the different artifacts as annotations. On UML models, we use the UMLsec profile for security annotations proposed by Jürjens [1]. Similar annotations are specified as Java annotations on the source code level and are also contained in the program model, like the `TSecrecy` annotation in Fig. 6 which relates to the *secrecy* value of the `<<critical>>` annotation in Fig. 2. Here, GRaViTY mainly allows the reuse of security requirements across the different artifacts. For example, as discussed in Sec. II-B, the UMLsec security annotations can be used to determine the sources and sinks of a secure data flow analysis.

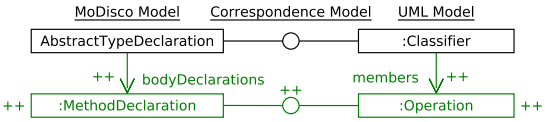
To keep the different artifacts consistent, we employ triple graph grammars (TGG) [21] for a bidirectional synchronization between the source code and the program model representation of Java programs [18] as well as UML models. Our implementation is based on the eMoflon graph transformation engine [22]. Among others, eMoflon allows the specification and execution of TGGs between models specified using the Eclipse Modeling Framework (EMF). While the UML models and the program model are specified using EMF, we have to parse the Java source code to create an EMF model from it. For this purpose, we are currently using MoDisco [23].

Fig. 7 shows two transformation rules from these TGGs, which translate a method declared by a type to a method definition in the program model or an operation in a UML class diagram respectively. Inbetween the models a correspondence model is built, that allows the synchronization of changes made on one of the two sides of the rule.

The single UML models are directly connected by inheritance relations, e.g., the `User` in the design model (Fig. 3) is a subtype of the `Patient` in the domain model (Fig. 2). This allows easy detection of changes that lead to inconsistencies, as the inheritance relations can be used as trace links, as demonstrated in Sec. II-B1. For this reason, the UMLsec tool is



(a) MoDisco Java Model ⇄ Program Model



(b) MoDisco Java Model ⇄ UML Class Diagram

Fig. 7: TGG Transformation Rules for Methods

integrated into GRaViTY. Unfortunately, unlike the mappings using TGGs, there is currently no automation in updating the different UML models.

Let us assume a change in the security knowledge and look at how the developed hospital system can be adapted to this change using the GRaViTY framework. Due to the introduction of the European General Data Protection Regulation (GDPR) [24], we got a stronger restriction in the ways how we have to deal with personal data. Before the GDPR became valid, it was legal to identify patients based on their names. This information has to be treated with more sensitivity now. This change in the security knowledge can, for example, be reflected in annotating the `Patient` in the domain model in Fig. 2 with the UMLsec stereotype `<<critical>>{secrecy={name:FullName}}` expressing that the access to this information is only allowed for legitimate cases. As this security annotation is inherited by the more concrete subtypes, the secure dependency check will fail after this change as there are no corresponding changes on the other elements. Accordingly, this gives a list of accesses to the developers, which have to be checked for this purpose. To do so, the developers have to look into the documentation and can follow the trace links generated by GRaViTY. Furthermore, they can use the TGGs to transfer the new security annotations into the code and re-execute the security analyses to get more detailed feedback about the compliance of the implementation.

To conclude, our TGGs provide an automated mechanism to preserve consistency between the three different program representations for managing evolving Java programs. As a result, we obtain a model-based framework for arbitrarily interleaving program evolution and maintenance steps. Furthermore, we can use this approach to also translate and synchronize security requirements of model elements between different system representations to execute sophisticated security checks on them as discussed in section II-B.

IV. Challenges to Overcome

During our work, we faced various challenges of which some have been solved by us, some have been circumvented by us, for some we have ideas on how to deal with them, and some are still open challenges. In this section, we are discussing the three most important challenges we faced.

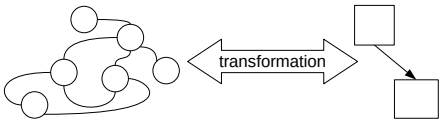


Fig. 8: Challenge A: Transformations between Models with Different Granularity

A. Transformation between Models with Different Granularity

Changes between synchronized UML models and code can easily be propagated, when they are on the same level of granularity, e.g., using the TGG-based approach presented in Sec. III. Models that are modeled by developers in early phases, e.g., the iTrust design model in Fig. 3, have a different granularity. Nevertheless, we have to be able to apply our synchronization approach also to those manually defined models. Accordingly, the first challenge is how to deal with such a different granularity, as shown in Fig. 8. At the development of GRaViTY, we faced this issue in three different variations.

a) Program Model: Our TGGs have been proven to be good in handling different granularity by not translating elements, e.g., all details from the method bodies available in the MoDisco model but not in the program model. Unfortunately, they cannot be used for creating structures that differ completely on the two sides. For this reason, we had to implement multiple preprocessing steps extending the different models with such structural information.

One example is the method representation as name, signature, and definition, shown in Fig. 6. While it is possible to create this structure using TGG rules by creating the whole structure when a method name is translated the first time and inserting afterward, this produces issues in the synchronization of changes. Let us assume that the `TMethodName` node in Fig. 6 has been created when the method in the class `EditOfficeVisit` has been translated and the other signature has been added afterward reusing this `TMethodName` node. In a refactoring, e.g., a pull-up method refactoring [19], we now delete the `TMethodDefinition` defined by `EditOfficeVisit` and are going to synchronize this change into the source code. To do this we have to undo all rule applications that lead to the creation of deleted nodes or edges. As the creation of the `TMethodName` node took place in the same rule application as the creation of the deleted `TMethodDefintion` node, it will be like this `TMethodName` node has never been created. This also makes the creation of the other `TMethodSignature` node invalid as its context does not exist anymore, leading to a situation in which no recovery is possible. To deal with this issue we defined a preprocessing which already creates the required structure on the side of the MoDisco model.

Unfortunately, the handling of such issues by preprocessing brings an additional level of complexity to the implementation and makes the synchronization more difficult as most preprocessings also need a postprocessing undoing them.

b) UML Models: To overcome the different granularity between the manually maintained UML models and the source code, we generate a UML class diagram on the granularity of the implementation, that is kept in sync with the implementation, into the implementation model. This generated model is initialized based on elements available in the implementation, architectural, and design model. Afterward, code stubs, as well as trace links, can be generated from this generated model. If one of the inheritances is lost, e.g., due to a deletion it has to be manually recreated by a developer. Additions in the implementation are automatically synchronized into the generated part of the implementation model.

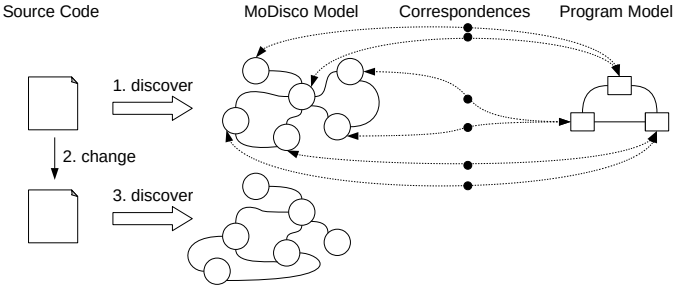


Fig. 9: Challenge B: Incremental Updates of ASTs

c) Security Requirements: The same as discussed before holds also for the different security requirements specified on every of these three system representations. While those security requirements should express the same assumption on every representation, this assumption should be expressed in an appropriate granularity on each representation: a more detailed specification is required on the source code level than in the domain or architectural model.

B. Incremental Updates of Abstract Syntax Trees (ASTs)

One of the biggest issues we faced is the loss of information, that was added manually or automatically to the created model, when the source code has to be parsed again. The same issue has been faced by representatives from industry, we talked to. This problem mainly covers the loss of generated or manually added annotations , such as the `TSecrecy` annotation in Fig. 6, and trace links to other artifacts, e.g., as part of the correspondence model built by the TGGs, as shown at the top of Fig. 9. Usually, the discovery of a model after changes on the code is executed from scratch resulting in an entirely new model. As the added annotations and trace links reside in the old model (as shown in Fig. 9) this results in the loss of all added information that has not been written into the source code. This means we would not be aware of the `TSecrecy` annotation on the method definition in Fig. 6 anymore and have no correspondences to the new model, as shown on the bottom of Fig. 9.

As there can be much information annotated to the models, if all this information would be written into the source code it could become unreadable. Also, this information is already stored at a different location and should not be duplicated.

In our case, the MoDisco framework builds an entirely new model each time, leading to the problem that the trace links of the TGG still point to the old model. The representatives

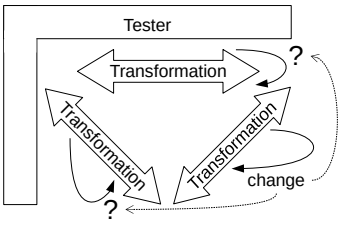


Fig. 10: Challenge C: Maintaining Networks of Transformations

from industry had the same problem with embedded C code. To deal with this issue, we generate and apply model patches to follow the source code changes. For this purpose we calculate the differences between the old and changed model using EMF Compare facing the following issues:

a) Scalability: At first EMF Compare behaved very well when we applied it to small artificial changes but did not scale on real changes. In our previous works, we built a test set of open-source projects with different sizes (between 5,800 and 200,000 lines of code) [2]. The creation of the program model takes between some seconds to some minutes dependent on the size of the program. For each program, we tried to calculate the differences between different versions of each program. While it took e.g. 7s to build a program model for JUnit version 3.8.1 (32,300 nodes in the MoDisco model) it took EMF Compare already 37s to calculate the differences between JUnit version 3.8.1 and 3.8.2. For our next bigger program GanttProject (146,000 nodes) the TGG is executed in 6s but EMF Compare did not even finish the comparison after 30 minutes.

b) Pseudo Differences: An additional issue we discovered is regarding the quality of the differences calculated by EMF Compare. We got many differences containing multiple changes that reverted themselves.

Helpful tool support going in the same direction, that could be utilized for this, might be a TGG based consistency check [25]. In this case, we have to specify a TGG transformation from the MoDisco meta-model to MoDisco that can be used to detect the differences between two MoDisco models.

However, the best would be an incremental parser for Java, that updates an initially created model each time it is executed on the same code again. Unfortunately, there are only a few works on this and none supports EMF.

C. Maintaining Networks of Transformations

The last challenge is regarding the maintenance of networks of transformations. According to Fig. 10, we have a network of transformations and are going to change one of the transformations. The open question is how to systematically derive the required changes on the other transformations and how a tester for detecting divergences has to look like.

While Fig. 5 shows three transformations between the different artifacts, by now we only implemented two of them. To be more precise, these are the MoDisco Java Model $\rightleftarrows$ UML Class Diagram and the MoDisco Java Model $\rightleftarrows$ Program Model transformations. Whenever we need a transformation between the UML Class Diagram and the Program Model we have to execute these two transformations in a row.

To speed up this process, we tried to generate a UML Class Diagram $\rightleftarrows$ Program Model transformation from the two specifications we already had. As the two transformations are translating the same elements from the MoDisco model, it should be straightforward to specify such a transformation. For example, as the two rules, in Fig. 7 both translate a `MethodDeclaration`, we can derive that we have to translate an `Operation` to a `TMethodDefiniton`. Unfortunately, while doing this, we had to learn that we have to resolve many inconsistencies first. Due to a very detailed test suite for the two transformations, this was surprising for us.

In this test suite, we created minimal examples for most Java language features to test the translation of these features. These features range from a simple class definition to exotic features such as the definition of an inner class inside of an anonymous class. All in all, our test suite contains 77 input models that can be given into the two transformations as well as the expected outcomes. All in all, we have 231 test cases in this test suite, 77 for the preprocessing common to the two transformations as well as 77 test cases for each of the transformations. Besides, we regularly execute the transformation on our test set of open-source projects, introduced in Sec. IV-B.

While testing the transformations we also had to deal with the model comparison problem discussed in Sec IV-B. Due to the high complexity of the models, it was also not possible to compare the generated model directly with an expected model without getting pseudo differences. Our solution to this was to specify essential expected patterns in Henshin rules [26] and to check whether the rules match as expected.

We had to learn that already a network with only two interacting transformations is hard to maintain. Thereby, we are in line with the observations of Stevens for bidirectional transformations [27].

As the test suite, which contains common input models to both transformations and expected outputs to the two transformations, was not enough to avoid an unnoticed divergence, we are currently thinking about other ways to test the transformation. One of the ideas we are currently thinking of is to specify the third missing transformation and to test by using a round-trip execution. To sum up, additional tool support for the maintenance of transformations is strongly required.

V. Related Work

In this section, we discuss how others dealt with the same challenges we faced in comparable approaches.

In the single underlying model approach (SUM), Atkinson et al. define a single model, that is able to express all information about the system [28]. Suitable views according to the current task are extracted from this model. The SUM is comparable to the different connected UML models of our approach, in which we integrate all design-time information. SUM supports an automated extraction of views that could be helpful in GRaViTY for manual edits of the generated parts of

the implementation model. While we support well known plain UML, SUM made many modifications to the UML to support all those kinds of different abstractions. Also, SUM does not provide an integration with the concrete implementation.

With VITRUVIUS Kramer et al. developed a SUM approach that also integrates Java source code [29]. Unlike our approach, the trace links to the model are written into the source code as annotations and might lead to unreadable source code, as discussed in Sec. IV-A.

On a very similar technical basis as our framework is the Codeling tool of Konersmann [30]. The idea of Codeling is the integration of architecture model information into the program code. Like our approach, Konersmann uses TGGs for model to model transformations at architecture extraction. In contrast to us, he is not continuously keeping the extracted models up to date but always writes all changes, made on an extracted model, back to the code. Every time an architectural view on the system is needed Codeling extracts it again. By doing so Konersmann is circumventing the challenge of incremental updates discussed in Sec. IV-B at the cost of massively increasing the code base with additional information.

Commercial tools like Enterprise Architect (EA) also provide round-trip engineering for UML models and Code [31]. The main limitation of these tools is the restriction to UML models very close to the code which eases the synchronization. While EA allows a translation from UML stereotypes to Java annotations, which could be used for transferring UMLsec annotations into the code, they do not support more complex information transfers.

While all approaches are dealing with the same challenges as us in similar ways, none of them provides the support to maintain security requirements on different artifacts in a sophisticated way and to check those security requirements in between the different artifacts.

VI. Conclusion

In this work, we presented the GRaViTY approach for model-driven development and maintenance of secure long-living systems. Based on GRaViTY, we elaborated on challenges we had to overcome for further automation in the development and verification of long-living systems using MDD.

The main challenges are in dealing with all the different levels of abstraction appearing in the development of systems and the synchronization of the single artifacts appearing. While we have been able to utilize recent developments from the model transformation domain for improving the synchronization between the different artifacts, we faced also challenges in maintaining those transformations ourselves.

To sum up, GRaViTY supports the model-driven development and maintenance of secure software systems by providing support to synchronize the different artifacts appearing during MDD as well as in the specification and reuse of security requirements in the execution of security checks for ensuring the security of the developed system.

References

[1] J. Jürjens, *Secure Systems Development with UML*. Springer, 2005, chinese translation: Tsinghua University Press, Beijing 2009.

[2] S. Peldszus, G. Kulcsár, M. Lochau, and S. Schulze, "Continuous Detection of Design Flaws in Evolving Object-Oriented Programs using Incremental Multi-pattern Matching," in *ASE*, 2016.

[3] J. Bürger, D. Strüber, S. Gärtner, T. Ruhroth, J. Jürjens, and K. Schneider, "A framework for semi-automated co-evolution of security knowledge and system models," *JSS*, vol. 139, 2018.

[4] S. Peldszus, K. Tuma, D. Strüber, J. Jürjens, and R. Scandariato, "Secure Data-Flow Compliance Checks between Models and Code based on Automated Mappings," in *MODELS*, 2019.

[5] D. L. Parnas, "Software Aging," in *ICSE*, 1994.

[6] "GRaViTY." [Online]. Available: http://gravity-tool.org

[7] T. Stahl, M. Voelter, and K. Czarnecki, *Model-driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

[8] B. Hailpern and P. Tarr, "Model-driven Development: The Good, the Bad, and the Ugly," *IBM Syst. J.*, vol. 45, no. 3, 2006.

[9] OMG, "UML Superstructure Specification," 2011.

[10] G. Wagner, *Information Management - An Introduction to Information Modeling and Databases*, 2019.

[11] UML-Diagrams, "Hospital management." [Online]. Available: https://www.uml-diagrams.org/examples/hospital-domain-diagram.html

[12] A. Meneely, B. Smith, and L. Williams, "iTrust Electronic Health Care System Case Study." [Online]. Available: http://agile.csc.ncsu.edu/iTrust

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Pearson, 1994.

[14] I. Christantoni, C. Biffi, D. Bonutto, and A. C. Sanz, "Vision pilots report," VisiOn Privacy Platform, Tech. Rep., 2017.

[15] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and R. Kamath, "CogniCrypt: Supporting Developers in using Cryptography," in *ASE*, 2017.

[16] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao, "A Static Analysis Framework For Detecting SQL Injection Vulnerabilities," in *COMPSAC*, 2007.

[17] S. Rasthofer, S. Arzt, and E. Bodden, "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks," in *NDSS*, 2014.

[18] S. Peldszus, G. Kulcsár, M. Lochau, and S. Schulze, "Incremental Co-evolution of Java Programs Based on Bidirectional Graph Transformation," in *PPPJ*, 2015.

[19] S. Peldszus, G. Kulcsár, and M. Lochau, "A Solution to the Java Refactoring Case Study using eMoflon," in *TTC*, 2015.

[20] S. Ruland, G. Kulcsár, E. Leblebici, S. Peldszus, and M. Lochau, "Controlling the Attack Surface of Object-Oriented Refactorings," in *FASE*, 2018.

[21] A. Schürr, "Specification of Graph Translators with Triple Graph Grammars," in *WG*, 1995.

[22] E. Leblebici, A. Anjorin, and A. Schürr, "Developing eMoflon with eMoflon," in *ICMT*, 2014.

[23] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering," in *ASE*, 2010.

[24] European Parliament and Council of the European Uninon, "Regulation (EU) 2016/679 – General Data Protection Regulation (GDPR)," in *Official Journal of the European Union*, 2016.

[25] E. Leblebici, "Inter-Model Consistency Checking and Restoration with Triple Graph Grammars," Ph.D. dissertation, 2018.

[26] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: Advanced Concepts and Tools for In-place Emf Model Transformations," in *MODELS*, 2010.

[27] P. Stevens, "Bidirectional Transformations in the Large," in *MODELS*, 2017.

[28] C. Atkinson, D. Stoll, and P. Bostan, "Orthographic Software Modeling: A Practical Approach to View-based Development," in *ENASE*. Springer, 2009, pp. 206–219.

[29] M. E. Kramer, E. Burger, and M. Langhammer, "View-centric Engineering with Synchronized Heterogeneous Models," in *VAO*, 2013.

[30] M. Konersmann, "Explicitly integrated architecture-an approach for integrating software architecture model information with program code," Ph.D. dissertation, 2018.

[31] "Enterprise Architect." [Online]. Available: www.sparxsystems.de